Compiler Project

The compiler project will be the major assignment for the quarter. This project will be divided in to four phases, where the due dates for each phase will be announced in class.

1. Scanner and Tokens

   At the end of this phase, a scanner will be produced that can read a C– file (.cm). The scanner will print each token along with the line number of the token. If the token is an identifier, the lexeme or string value of the identifier is printed; or, if the token is a number, the value of the number is printed.

   The scanner, the parser, and the code generator will need to use a standard set of constants. To make this as standardized as possible, an interface will be provided that implements the following as public integer constants.

   EOF, ERROR, ELSE, IF, INT, RETURN, VOID, WHILE, PLUS, MINUS, MULT, DIV, LS, LEQ, GT, GEQ, EQ, NEQ, ASSIGN, SEMI, COMMA, LPAREN, RPAREN, LBRACKET, RBRACKET, LBRACE, RBRACE, READ, WRITE, NUMBER, ID, PROGRAM, DECLARATION, VARIABLE, ARRAY, FUNCTION, EXPRESSION, CALL, COMPOUND, TYPE_SPECIFIER, PARAMETER_LIST, PARAMETER, STATEMENT_LIST, STATEMENT, and ARGUMENTS.

   The scanner and the tokens will be objects of separate classes. The scanner should have a constructor that takes a parameter that represents the file to be scanned. For example, it may be a `File` object or an object of a separate class that represents the file. You will also need a separate driver program that will contain the main method. This testing program can be used throughout the project, and it does not need to implement a GUI. For the first phase, the testing program gets a file name from the user, opens the file in a manner compatible with the scanner, instantiates a scanner object, and repeatedly calls the method to get a token until the EOF token is obtained. The testing program prints each token as it is obtained from the scanner.

2. Parser and Syntax Tree

   At the end of this phase, a recursive descent parser will be written that accepts a syntactically correct C– (.cm) program and reports an error for a syntactically incorrect program. The parser does not have to do any error correction, it can report the first error it finds and then quit. If you examine the grammar for a C– program in Appendix A, you will find a BNF grammar for C–. This has to be converted to an EBNF grammar with the left-recursion removed before it can be used with a recursive descent parser. In addition, I am adding two statements to the grammar: one that will read a value and the other will write an expression.

   As the program is parsed, it builds an abstract syntax tree. If the program is successfully parsed, a copy of the syntax tree is printed out for this phase. I have a document that gives you the specifications for the syntax tree along with a sample program and the tree that is